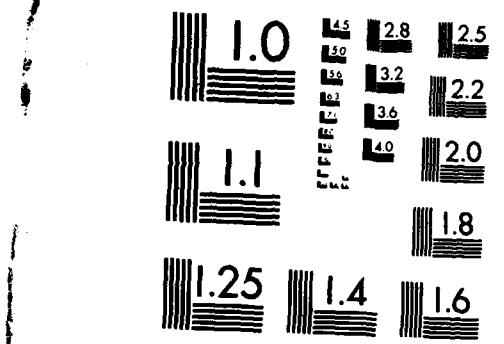
AD-A137 108 ADAREL: A RELATIONAL EXTENSION OF ADA(U) UNIVERSITY OF SOUTHERN CALIFORNIA LOS ANGELES DEPT OF COMPUTER SCIENCE E HOROWITZ ET AL. 1983 AFOSR-TR-83-1309 1/1

UNCLASSIFIED AFOSR-82-0232 F/G 9/2 NL



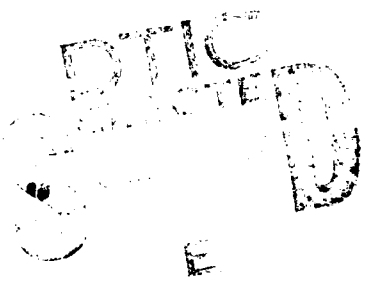
AdaRel: A Relational Extension of Ada

Ellis Horowitz
Alfons Kemper

COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF SOUTHERN CALIFORNIA
LOS ANGELES, CALIFORNIA 90089-0782

AdaRel: A Relational Extension of Ada

Ellis Horowitz
Alfons Kemper



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S): AFOSR-TR- 83-1309
6a. NAME OF PERFORMING ORGANIZATION: University of Southern California; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: Air Force Office of Scientific Research
6c. ADDRESS (City, State and ZIP Code): Computer Science Department, University Park, Los Angeles CA 90089-0782	7b. ADDRESS (City, State and ZIP Code): Directorate of Mathematical & Information Sciences, Bolling AFB DC 20332
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: AFOSR; 8b. OFFICE SYMBOL (If applicable): NM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: AFOSR-82-0232
8c. ADDRESS (City, State and ZIP Code): Bolling AFB DC 20332	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
61102F	2304	A2	

11. TITLE (Include Security Classification): AdaRel: A Relational Extension of Ada

12. PERSONAL AUTHOR(S): Ellis Horowitz and Alfons Kemper

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	1983	25

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

In this paper the authors extend Ada to facilitate the programming of data-intensive applications. The language extensions are based upon the relational data model. The system is interfaced to a relational database management system via a new Ada type relation. The language includes basic operations on relations, commonly available in database query languages, like retrieval of data, update of tuples as well as high-level operators to combine relations to form new ones. The authors show how Ada exception handling is naturally extended to allow integrity control of the relations. In addition the authors discuss language features that enable the sharing of data among several users. Concluding the paper the authors give an extensive example application to demonstrate the power of their proposed language extensions.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☒ DTIC USERS ☐	UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
Dr. Robert N. Buchal	(202) 767- 4939	NM

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

AdaRel: A Relational Extension of Ada

Ellis Horowitz and Alfons Kemper
Computer Science Department
University of Southern California
Los Angeles, California 90089

This work has been supported by the Air Force Office of Scientific Research under Grant no. AFOSR–82-0232

Abstract

In this paper we extend Ada to facilitate the programming of data-intensive applications. The language extensions are based upon the relational data model. The system is interfaced to a relational database management system via a new Ada type *relation*. The language includes basic operations on relations, commonly available in database query languages, like retrieval of data, update of tuples as well as high-level operators to combine relations to form new ones. We show how Ada exception handling is naturally extended to allow integrity control of the relations. In addition we discuss language features that enable the sharing of data among several users. Concluding the paper we give an extensive example application to demonstrate the power of our proposed language extensions.

Table of Contents

List of Figures

1. Introduction

In the last decade it has been realized [4, 9, 11] that there is a growing need for providing good, user-friendly software systems to support the development of data intensive application. This led to the design and implementation of several database management systems like System R, Ingres [20] as well as more high-level application generators [6, 10, 12]. Application generators are high-level non-procedural programming systems that interface to a database management system [8]. But, as we pointed out in an earlier paper [8], application generators just like database management systems lack the computational flexibility of general purpose programming languages, which is often required to code interactive database applications.

Therefore, in this paper we describe AdaRel, an integration of database constructs in the general purpose programming language Ada [1]. In addition to the flexibility of a general purpose programming language this will gain us the expressive power of a high-level relational data manipulation language. But by no means do we change the Ada base language. Moreover, our language extensions comply with the major design goals of Ada, in particular strong typing. In our approach we decided to interface Ada to a relational DBMS. This was done because it is widely agreed [21] that the relational datamodel is the most user-friendly of the three major data models: relational, hierarchical, network. We believe that it is better to base our design on a data model that has been readily implemented and tested rather than basing it on an experimental data model, as was done in the design of Adaplex. Adaplex [19] is an embedding of the *functional* database language Daplex [17] in Ada. Also we think that relations are a more natural extension of the existing Ada concepts, i.e. records, than the entities and functions of the functional data model.

There are a few other proposals of relational extensions for other programming languages. Pascal/R [16, 15] and THESEUS [18] are embeddings of relational database features in the existing languages Pascal and Euclid. The basic relational operators in these two languages are very similar to the ones we incorporated in Ada. But neither of these two language proposals has addressed exception handling, data abstraction, integrity constraint, or concurrency control issues. Plain [22] and Rigel [13] are newly designed languages with an emphasis on database features. Since they are completely new languages, i.e. they are not based on a widely known baselanguage, these two designs are of a rather experimental nature.

The rest of this paper is organized as follows. We will first describe the new Ada type relation and the basic operations on relations. Then in section 3 we discuss language features that allow the sharing of data among several users. For this purpose we have to introduce concurrency control mechanisms in the language which guarantee mutual exclusive access to the data stored in the database. Section 4 describes a natural extension of the Ada exception handling for the relational operations as well as for declaration

and enforcement of integrity constraints on relations. Finally in section 5 we present a non-trivial example application implemented in our proposed language extension.

2. Designing Database Capabilities into Ada

In this section we will show how database features can be integrated into Ada. We will first introduce a new data type *relation*. Then we describe the basic operations that are available on this new data type.

2.1. Declaration and Initialization of Relations

A relation in AdaRel is defined very similar to an Ada record. The only exceptions are:

1. A relation cannot have a variant part.

2. A relation must have specified one or more key fields which have to be unique for each tuple in that relation, i.e. no two (different) tuples can agree on the key fields.

The syntax for type declarations of relations is as follows:

```
<relation-type-definition>::=
type <rel-name> is
        relation (key <attr-name> {,<attr-name>}) of
                <attr-name>:<type-name> [:=<null-value>];
                {<attr-name>:<type-name> [:=<null-value>];}
        end relation
```

<rel-name> and <attr-name> have to be valid Ada identifiers. The key fields have to be defined inside the relation declaration. The possible types of the fields are any numerical type that Ada allows, enumerated types, and strings. Composite types, such as records and arrays, or pointer types are not allowed. <null-value> denotes a value of the type <type-name> chosen by the programmer, such that when a tuple is assigned to a relation, but this particular field does not have a value, the <null-value> is automatically assigned.

A relation variable is a set of tuples (essentially records), corresponding to the rows in the relation. Each tuple consists of the fields specified in the type declaration of that relation. Thus we could view a relation as a multidimensional record. An example of a declaration of a relation, in this case a relation to store US cities, is given as:

```
type US_STATES is (CA, ... ,NY);  --list all states

type CITIES is
        relation (key NAME) of
                NAME:string;
                STATE:US_STATES;
                POPULATION:integer;
        end relation
```

Now we can declare two relation variables of type CITIES as below:

```
ALL_CITIES,BIG_CITIES:CITIES;
```

To initialize relations we can naturally extend the Ada record assignment feature:

```
BIG_CITIES:=<("L.A.",CA,3_000_000),("N.Y.",NY,9_000_000)>;
```

This assignment will assign the two tuples ("L.A.",CA,3_000_000) and ("N.Y.",NY,9_000_000) to the relation variable BIG_CITIES.

Access to fields of a tuple in a relation is similar to the access of fields in a record. The difference now is that there are (possibly) more than one tuple in a relation. Therefore we have to uniquely identify the desired tuple. This is done as shown in the example below where we assign the population of L.A. to the integer variable SIZE.

```
SIZE:integer;
        ...
SIZE:=BIG_CITIES[NAME="L.A."].POPULATION;
```

This will assign 3000000 to the integer variable SIZE. The tuple selector(s) inside the square brackets must uniquely identify one tuple. In this example we specified the key field, which must, by definition, uniquely identify one tuple. In general we are allowed to specify any number of fields of a relation, whether they are key fields or not, as long as they identify one unique tuple. For example we could retrieve the name of the city that has 9 million inhabitants and is located in the state NY as:

```
BIG_CITIES[STATE=NY,POPULATION=9000000].NAME
```

If more than one tuple satisfies the specification inside the square brackets the predefined exception AMBIGUOUS_TUPLE_SELECTOR will be raised. For a discussion of exception handling in the context of relations see section 4. In the next subsection we will show how to retrieve more than one tuple at a time from a relation by specifying selection criteria that are satisfied by a subset of the tuples in the relation.

Since we want to be able to read and modify existing *external relations* we need to provide an interface between Ada and a database management system. For this we introduce the predefined function EXTERNAL_REL. This lets the programmer associate an externally stored relation with an internally declared relation, i.e. the system will look up this particular relation in the corresponding DBMS. Of course, this relation must have been defined there prior to the use of it in the Ada program. Also the fields of the relation have to agree with the type definition given in the Ada program. As an example let's say the programmer wants to make use of an external database of all US cities that is stored in the DBMS in the relation US_CITIES. In the program one would declare this relation as follows:

```
CITY_REL:CITIES:=EXTERNAL_REL("US_CITIES");
```

This means that CITY_REL is now a relation variable of type CITIES. During elaboration of the declaration the value of this variable becomes all the tuples that are stored in the corresponding relation US_CITIES in the DBMS. Of course the tuples needn't get physically copied, rather a pointer to the

external relation is maintained.

2.2. Operations on Relations

In order to retrieve one tuple at a time one has to define a record which is of the type of the underlying relation. Thus one might define:

```
type ONE_CITY is record of
                NAME:string;
                POPULATION:integer;
                STATE:string;
    end record
```

In order to avoid unnecessary repetition a new keyword recordtype is introduced. Using th ... can define the type ONE_CITY as shown below:

```
type ONE_CITY is recordtype CITIES;
```

This declaration is equivalent to the complete declaration where we have to spell out all the fields of the correponding relation. Now we can assign to a variable LA of type ONE_CITY one whole tuple from the corresponding relation as follows:

```
LA:ONE_CITY;
   ...
LA:=BIG_CITIES[NAME="L.A."];
```

and to retrieve some field from this record we merely write

```
SIZE:=LA.POPULATION;
```

Frequently we need to traverse through all the tuples in a relation that fulfill a certain boolean expression. This can be done with the explicit loop construct which has the form described in Fig. 2-1.

```
for <record-var> in <rel-name> [where <screening-cond>]
        loop
          ...
        end loop
```

An example of the usage is given as:

```
ONE_BY_ONE:ONE_CITY;
        ...
for ONE_BY_ONE in BIG_CITIES where ONE_BY_ONE.STATE=NY
        loop
        ...     --fields in record ONE_BY_ONE
        ...     --can now be accessed
        end loop
```

Figure 2-1: Syntax and Example of the Loop Construct

The <screening-cond> following the keyword where can be any boolean expression whose scope includes the fields of the relation. For each tuple of the relation in turn, this boolean expression is evaluated. If it

evaluates to true the corresponding tuple is assigned to the record <record-var> and, subsequently, can be accessed inside the loop ... end loop delimiters. In the above example in Fig. 2-1 all the tuples of the relation BIG_CITIES whose STATE field equals NY are assigned to the record ONE_BY_ONE and are then processed within the loop delimiters.

Surely we need to be able to update existing relations, i.e. change values of fields of certain tuples. One way to do this is by explicitly assigning the new value, e.g.

```
<rel-name>'['<tuple-selector>']'[1].<attr-name>:=<value>
```

An example of such an update is given below:

```
BIG_CITIES[NAME="L.A."].population:=2_000_000;
```

If more than one field is to be changed it might be more appropriate to use the following language feature that makes use of a record <rec-name> whose type corresponds to the relation <rel-name>.

```
update <rel-name>'['<tuple-selector>']' to <rec-name>;
```

Thus if we had a record, say NEW_LA of type ONE_CITY, with the updated value for population we could write:

```
update BIG_CITIES[NAME="L.A."] to NEW_LA;
```

If we have to update the information in all tuples that fulfill a certain screening condition we can make use of the loop construct. Let's say we want to make the update of increasing the population of all cities in California by 10 percent. This can be done as follows:

```
for ONE_BY_ONE in US_CITIES where ONE_BY_ONE.STATE=CA
loop
   US_CITIES[ONE_BY_ONE].POPULATION:=ONE_BY_ONE.POPULATION*1.1;
end loop
```

In this case the record ONE_BY_ONE is used as the tuple selector inside the square brackets.

2.3. Relational Operators to Generate New Relations

In order to generate new relations from existing ones we introduce the following traditional relational operators: select, project, join, union, difference, intersect. The syntax of these operators is described in Fig. 2-2. The screening condition (for the select operator) is some boolean expression over the fields of the relation. Its semantics is the same as for the loop statement which was described in Fig. 2-1. Note that for the union operator it is possible to have a record as an argument. This provides for insertion of new tuples into a relation. Of course, the record has to be of the type corresponding to the relation. Also the difference operator can have a record variable as a second argument to allow deletion of tuples from a relation.

---

[1] In this case the square brackets are part of the language feature and do not denote an alternative choice in the BNF. This is indicated by the single quote.

```
select:
        <rel-name> := select <rel-name> where <screening condition>
project:
        <rel-name> := project <rel-name> on (<attr-name>{,<attr-name>})
join:
        <rel-name> := join (<rel-name>,<rel-name>) on
          (<attr-name> bool-op <attr-name> {;<attr-name> bool-op <attr-name>})
union:
        <rel-name> := <rel-name>|<rec-name> union <rel-name>|<rec-name>
difference:
        <rel-name> := <rel-name> difference <rel-name>|<rec-name>
intersect:
        <rel-name> := <rel-name> intersect <rel-name>
```

Figure 2-2: Syntax of the Relational Operators

The use of the operators select, union, difference, and intersect results in a new relation which is of the same type as the relations to which the operator is applied. But the operators project and join result in a new relation type. Since one of the design principles of Ada is strong typing we have to declare each relation before it can be used in the program. It would be too tedious for the programmer to give the complete table definition of a relation that is generated by one of the two operators join or project. For this reason we introduce the jointype and projecttype features which are analogous to the earlier introduced recordtype construct. The syntax is:

```
type <rel-name> is jointype (<rel-name>,<rel-name>)
        on (<attr-name>,<attr-name> {;<attr-name>,<attr-name>})
```

and

```
type <rel-name> is projecttype <relname>
        on (<attr-name>{,<attr-name>})
```

These features let the programmer define a new relation type which is derived from previously defined relation types and the respective operation on them, i.e. join or project. The system will automatically determine the attribute names and types from the relation types over which jointype respectively projecttype is defined.

Now let us give some examples of the use of these relational operators. The assignment

```
BIG_CITIES:=select US_CITIES where POPULATION > 1000000;
```

assigns all those tuples of the relation US_CITIES to the relation BIG_CITIES that have a population of more than one million. We note that BIG_CITIES and US_CITIES are of the same type.

If we want to create a (temporary) relation that contains just the names of the big cities we can use the project operator as follows:

```
JUST_CITY_NAMES:projecttype CITIES on (NAME);
                ...
```

```
JUST_CITY_NAMES:=project BIG_CITIES on (NAME);
```

Now JUST_CITY_NAMES is a relation with just one field, namely NAME. The type of the relation JUST_CITY_NAMES can be deduced from the project operation, therefore it has been declared using projecttype. For efficiency reasons, the system will not necessarily physically construct this relation but rather maintain pointers to the appropriate fields in the BIG_CITIES relation. This means that JUST_CITY_NAMES is a view of the relation BIG_CITIES in the sense that all fields are logically blanked out except for the "NAME" field.

The union operator can be used to create one relation that contains all the tuples of the two relations on which this operator is applied. Let SMALL_CITIES as well as ALL_CITIES be declared as a relation of type CITIES. Then

```
ALL_CITIES:=BIG_CITIES union SMALL_CITIES;
```

results in a relation ALL_CITIES which contains the tuples for small cities as well as for large cities. A problem arises if both relations contain a tuple with identical key fields, in this case NAME. If that happens the system will have to raise an exception. Exception handling will be discussed in a later section of this paper.

Let us now declare a new relation type to demonstrate the use of the join operator.

```
type DENSITY_INFO is
        relation (key NAME) of
                NAME:string;
                DENSITY:float;
        end relation
```

Let CITY_DENSITY be a relation of type DENSITY_INFO, i.e.

```
CITY_DENSITY:DENSITY_INFO;
```

If we want to have all the information about the cities in one relation with the fields NAME, STATE, POPULATION, and DENSITY we have to join the two relations on identical NAME fields.

```
ALL_INFO:jointype (US_CITIES,CITY_DENSITY) on (NAME,NAME);
                    ...
ALL_INFO:=join (US_CITIES,CITY_DENSITY) on (NAME=NAME);
```

In this case the two relations happen to be joined on the key fields of both of them. But this is not required as any fields are allowed as long as they are of compatible types.

The semantics of the difference and intersect operators is obvious.

We feel that relations provide an elegant means to handle large amounts of data in a program. Storing information in relations frees the user from deciding and/or knowing the physical representation of the data that he wants to access or modify. All the programmer has to do is specify, via appropriate tuple

selectors, which tuples he wants to access. Then it is up to the system to find a way to physically retrieve the tuple. This makes the writing of programs substantially easier. For the same reason we feel that the readability of programs is enhanced if the data is stored in relational form.

3. Concurrency Control in AdaRel

Usually databases are accessed and modified by more than one user (or application program). An example of this is an airline reservation system where the database of available seats can be accessed by all connected travel agencies. Another example is a computerized banking system where the account of a particular customer can be accessed from all different branches of that bank. This necessitates the concurrent access to the data stored in a centralized database. But the uncontrolled simultaneous access and modification of data by two different (or two parallel executions of the same) programs may generate undesired results. For example it could happen that the same airline seat gets sold twice by two different travel agencies who happen to access the database simultaneously and both find that the particular seat is still available.

Figure 3-1 shows an example of two AdaRel program fragments that could possibly run concurrently in a banking system. If these programs were run concurrently without any concurrency control such as locking the databases involved the result of the function SUMUP could be wrong. Consider an example of an account relation as in Figure 3-1. As indicated by the name the procedure transfer handles (very simplified) transfers of money between accounts within the same bank (relation). The amount to be transfered gets first deducted from A's account and is then added to the BALANCE of B's account.

Consider the example relation LA_ACCTS of type ACCOUNTS as illustrated in Figure 3-1. Assume the procedure TRANSFER is called as TRANSFER("SMITH","MAYER", 50,LA_ACCTS) and the execution of this procedure is parallel to the execution of the function SUMUP. The function SUMUP totals the amount of money in the bank, i.e. the sum of all BALANCEs of all accounts. Thus the call SUMUP(LA_ACCTS) should return the amount of money in all accounts in this relation. But if SUMUP(LA_ACCTS) is executed at time T1, i.e. in the middle of the transfer transaction the returned result will be wrong. At time T1 the procedure TRANSFER has deducted $50.00 from SMITH's account but not yet increased the BALANCE of MAYER's account. Therefore the result of SUMUP is $550.00 instead of the correct $600.00.

3.1. Locking and Unlocking Relations

In a distributed database system we need to assure that no two processes access the same two data items concurrently. In existing distributed database systems this is usually achieved by "locking" all the data items that are involved in a transaction and then "unlocking" them after the transaction has been successfully completed. If we included language constructs to lock and unlock relations in AdaRel we

```
                ...
type MONEY is delta 0.01 range -10E8..+10E8;

type ACCOUNTS is relation(key NAME) of
        NAME:string(20);
        BALANCE:MONEY;
        STATUS:(savings,checking);
        BRANCH:string(20);
end relation;

P1:

procedure TRANSFER(A,B:string(20),AMT:MONEY;inout ACCT:ACCOUNTS) is;
        begin
                ...
                ...
        ACCT[NAME=A].BALANCE:=ACCT[NAME=A].BALANCE-AMT;

// Time T1 --->

        ACCT[NAME=B].BALANCE:=ACCT[NAME=B].BALANCE+AMT;
                ...
                ...
        end procedure;


P2:

function SUMUP(ACCT:ACCOUNTS) return MONEY is
SUM:MONEY:=0.00;
        begin
                ...
                ...
                for A in ACCT loop              //A appropriately declared
                        SUM:=SUM+A.BALANCE;  //as record elsewhere
                end loop;
                return SUM;
                ...
                ...
        end function;
```

Figure 3-1: Two Possibly Interfering AdaRel Programs Using the Same Relation

LA_ACCTS	NAME	BALANCE	STATUS	BRANCH
	SMITH	100.00	...	...
	MILLER	200.00	...	...
	MAYER	300.00	...	...

Figure 3-2: Example of an Accounts Relation

would rewrite the two subroutines of Figure 3-1 as illustrated in Figure 3-3.

```
procedure TRANSFER(A,B:string(20),AMT:MONEY;inout ACCT:ACCOUNTS) is;
        begin
        lock relation ACCT;
                ...
                ...
        ACCT[NAME=A].BALANCE:=ACCT[NAME=A].BALANCE-AMT;

        ACCT[NAME=B].BALANCE:=ACCT[NAME=B].BALANCE+AMT;
                ...
                ...
        unlock relation ACCT;
        end procedure;


function SUMUP(ACCT:ACCOUNTS) return MONEY is
SUM:MONEY:=0.00;
        begin
                lock relation ACCT;
                ...
                ...
                for A in ACCT loop
                        SUM:=SUM+A.BALANCE;
                end loop;
                return SUM;
                ...
                ...
                unlock relation ACCT;
        end function;
```

Figure 3-3: Two AdaRel Subroutines of Fig. 3-1 Rewritten Using Locks

This approach suffers from several drawbacks:

- The locking and unlocking of relations explicitly in the program is very clumsy. It should be automated. The programmer should be able to just specify where a transaction begins that cannot be interrupted and then the system should automatically lock (and then unlock) all relations involved in the transaction.

- We should provide different locks for reading data and writing data because it is legitimate to have more than one reader process at a time accessing the data. But a writer process needs to have an exclusive lock on the data. This provides a possibly higher degree of parallelism.

- It is not always efficient to lock a complete relation. If a transaction consists of merely updating one record of a large relation it is more appropriate to just lock this one record so that other application programs can still access the rest of the relation.

In conclusion the locking and unlocking of data items should not be explicitly done by the user. Chamberlain et al. pointed out that the locking protocol should be invisible to the programmer [3]. The system should provide a so called "locking manager" which automates this task.

3.2. Atomic Transactions

Ullman [21] defines a transaction as one execution of a particular database program. But for our purposes it is not feasible to view a whole program as one non-interruptable transaction for which the "locking manager" would aquire locks on the data involved. This would, for most applications, outrule any parallelism and therefore make sharing of data impossible. Consider for example the following loop (fragment) which might very well be the skeleton of an airline reservation system:

```
while input /= "end of day" loop
        ...
        ...     // process input using
        ...     // the airline reservation
        ...     // database
        ...
end loop
```

If we considered this program as an uninterruptable processing unit we could not have more than one program running simultaneously. The locking manager would once and for all aquire locks for the data in the airline reservation database and not release it before the program reads the input "end of day".

This led to the notion of "atomic transaction" in database languages, such as SEQUEL [2] and Adaplex [19]. A single program can now consist of several atomic transactions. An atomic transaction is the processing unit that cannot be interrupted by parallel processes which need access to the same data items. The extent of an atomic transaction is explicitly defined by the programmer. *In AdaRel we will introduce* the keywords

```
atomic transaction
        ...
end atomic transaction
```

to delimit the extent of an atomic processing unit. The (system) default, i.e. if the programmer doesn't specify anything, is that each statement in a AdaRel program is an atomic transaction by itself. Now the above program fragment of an airline reservation system would typically have the form

```
while input /= "end of day" loop
        atomic transaction
        ...     // process input using
        ...     // the airline reservation
        ...     // database
        end atomic transaction
end loop
```

This means that the processing of each new input is treated as an atomic transaction. After a particular input is processed all locks on data items are automatically released by the locking manager such that other processes can access and modify the same set of data.

Using the notion of atomic transaction the subroutine TRANSFER and SUMUP are coded as shown in Figure 3-4. We note that the (automatic) locking manager has to obtain just read-only locks for the

```
procedure TRANSFER(A,B:string(20),AMT:MONEY;inout ACCT:ACCOUNTS) is;
        begin
                ...
                ...
        atomic transaction

          ACCT[NAME=A].BALANCE:=ACCT[NAME=A].BALANCE-AMT;

          ACCT[NAME=B].BALANCE:=ACCT[NAME=B].BALANCE+AMT;

        end atomic transaction
                ...
                ...
        end procedure;


function SUMUP(ACCT:ACCOUNTS) return MONEY is
SUM:MONEY:=0.00;
        begin
                ...
                ...
            atomic transaction
                for A in ACCT loop                  //A appropriately declared
                        SUM:=SUM+A.BALANCE;  //as record elsewhere
                end loop;
            end atomic transaction
            return SUM;
                ...
                ...
        end function;
```

Figure 3-4: Two Subroutines of Fig. 3-1 Recoded Using Atomic Transactions

SUMUP function.

It would have been wrong to code the function SUMUP as in Figure 3-5. This would lead to the same error described for the subroutines given in Figure 3-1. Furthermore this version of the function SUMUP would have been terribly inefficient. The locking manager would have to acquire a lock for the relation ACCT every time the body of the loop is executed, i.e. for each tuple in the relation and then release this lock each time. For a sufficiently large relation ACCT the overhead for this would be quite high.

We conclude that even though the programmer does not have to explicitly lock the relations involved in a database transaction he still has the responsibility to identify the atomic transactions of the program in a correct and hopefully efficient way.

```
function SUMUP(ACCT:ACCOUNTS) return MONEY is
SUM:MONEY:=0.00;
        begin
                ...
                ...
                for A in ACCT loop                  //A appropriately declared
                    atomic transaction

                        SUM:=SUM+A.BALANCE;  //as record elsewhere

                    end atomic transaction
                end loop;
                return SUM;
                ...
                ...
        end function;
```

Figure 3-5: Erroneous Specification of Atomic Transactions in Function SUMUP

3.3. Error Recovery

In this subsection we will consider the issue of error recovery, i.e. restoring the database to a consistent state after a program crash. To demonstrate that this issue does not just involve implementation details consider the following AdaRel example program in Figure 3-6.

```
procedure ADD_INTEREST(RATE:float;inout ACCTS:ACCOUNTS) is
        begin
           atomic transaction
                for A in ACCTS where BALANCE > 0.0 loop
                        A.BALANCE := A.BALANCE * (1+RATE);
                end loop
           end atomic transaction
        end procedure
```

Figure 3-6: Procedure to Add Interest to Accounts

If the procedure crashes for some (unknown) reason in the middle of the execution of the loop the data in the relation ACCTS will be left in an inconsistent state. Even worse, the data is in a state from which it is not possible to restore it to the state it was in at the beginning of the transaction. For some accounts the interest has already been added to the balance, for others it hasn't. But there is no way of knowing since the AdaRel loop does not make any assumptions on the order in which the data in the relation is processed.

The *atomic transaction* concept can be used for error recovery resulting from situations as demonstrated above. We require that all data modifications within an atomic transaction are first done on a back-up copy of the database. Thus the original data is left in the same state it was in at the beginning of the atomic transaction. The changes to the data become permanent only after the atomic transaction has been succesfully terminated. If it abnormally terminates the exception ATOMIC_ERROR is raised

and the data is left in the original state so that the user can restart the transaction starting from the initial state.

4. Exception Handling and Integrity Constraints

In this section we will discuss exception handling and integrity control in the context of operations on relations. As we will see both concepts are natural extensions of the already available exception handling mechanism of Ada.

4.1. Exception Handling

In this subsection we will investigate how the Ada exception handling facility can naturally be extended to deal with the incorporated database and application generator facilities. In addition to the existing built-in Ada exceptions we add the following predefined exceptions to the language:

SAME_KEY_TWICE
: this exception will be raised if the programmer attempts to have two different tuples in a database relation with identical key field(s).

AMBIGUOUS_TUPLE_SELECTOR
: whenever the specified tuple selector (used to access some particular tuple) does not uniquely identify one tuple in the relation this exception is raised.

NO_SUCH_TUPLE
: is raised whenever the access to a tuple fails because the relation in question does not contain any tuple of the particular form.

KEY_FIELD_UNSPECIFIED
: raised when the user tries to insert some tuple into a relation for which one (or more of the) key field(s) are unspecified.

DIFFERENT_TYPE_JOIN
: if a join is attempted on fields whose types are not compatible this exception will be raised.

INTEGRITY_VIOLATION
: this predefined exception is used to declare integrity constraints on relations. The use of this will be explained in the next subsection.

In adition to the predefined exceptions the programmer can declare his/her own exceptions just like in conventional Ada [1].

In Fig. 4-1 we show an example program that makes use of the built-in exception SAME_KEY_TWICE. This interactive program first determines what kind of operation the user wishes to perform, i.e. insertion, deletion, ... We merely provide the implementation for insertion of a new tuple in the relation. The new city record is read in and then appended to the relation BIG_CITIES via the union statement. This is done inside an atomic transaction block. If it turns out that the particular city

```
...
begin
     declare NEW_CITY:ONE_CITY;  --ONE_CITY was declared earlier
     ...
     loop
       PROMPT(USER_REQUEST);  --function to read new request
       case USER_REQUEST is
            when DONE =>
                   exit;  --finished for the day
            when DELETE =>
                   ...    --code for deletion of tuple
                   ...
            when INSERT =>
               begin
                    atomic transaction
                       READ_IN(NEW_CITY);  --reads in a new record
                       BIG_CITIES:=BIG_CITIES union NEW_CITY;
                    end atomic transaction
                    exception
                       when SAME_KEY_TWICE =>
                            update BIG_CITIES[NAME=NEW_CITY.NAME]
                            to NEW_CITY;

               end  --INSERT

                    ...
                    ...

     end loop;
end
```

Figure 4-1: Example Program Using the Exception SAME_KEY_TWICE

existed already in the relation the program would automatically exit from the atomic transaction block and raise the exception SAME_KEY_TWICE after restoring the relation BIG_CITIES to its consistent state prior to the execution of the atomic transaction, in this case prior to the union statement. The exception handler that we provided specifies that instead of the union operation an update on the relation is to be performed, i.e. the existing tuple for the particular city becomes updated to the fields of the new city record NEW_CITY. This example program demonstrates how the atomic transaction concept together with the extended Ada exception handling facility can help to keep the database relations in a consistent state.

4.2. Integrity Control

Integrity constraints are programmer defined predicates on relations which have to be satisfied at all times. An example of such a construct might be that "the sum of the population of all cities in the relation US_CITIES cannot exceed the total population of the United States". If at some point in time this constraint is violated, due to insertions and/or updates of the relation, the system will raise the

exception INTEGRITY_VIOLATION. Integrity constraints can be defined for types of relations as well as relation variables. If an integrity constraint is defined for a relation type this constraint will be enforced for all variables of that type. In the case that the constraint is defined for a relation variable it will be enforced only for that variable and not for other variables of the same type. Integrity constraints can be declared at any place where a variable or subroutine declaration can take place in Ada. The usual visibility rules of Ada apply to integrity declarations, that is an integrity constraint will be checked for as long as it is visible in the program block.

The syntax of integrity declarations is outlined below:

```
integrity <constraint-name> (<param-list>) is
<declarative part>
begin
        <constraint body>
end integrity
```

The <param-list> contains the list of relation variables and/or relation types for which this integrity constraint has to be checked. An example of such a declaration is given in Fig. 4-2. In this example the predefined exception INTEGRITY_VIOLATION is raised whenever the sum of the population of all cities in one particular state exceeds the total population of that state.

```
type US_STATES is (CA,MA,NC,.....,NY);
                  ...

integrity POPULATION_CHECK(US_CITIES:CITIES) is
TOTAL:integer;
TEMP:CITIES;
begin
    for ST in US_STATES'FIRST..US_STATES'LAST loop
          TEMP:=select US_CITIES where STATE=ST;
          TOTAL:=POP_SUM(TEMP),   --assume this function
          case ST is
               when CA  =>if TOTAL>20000000 then
                                        raise INTEGRITY_VIOLATION;
                        ...
                        ...
               when NY =>if TOTAL>40000000 then
                                        raise INTEGRITY_VIOLATION;
          end case;
    end loop;
end   --POPULATION_CHECK
```

Figure 4-2: An Example of a Constraint Definition

The integrity constraint in Fig. 4-2 is defined for the variable US_CITIES. An example of an integrity declaration for a relation type is given in Figure US_POP. Now whenever the relation variable US_CITIES is altered via updates or insertions the integrity constraint POPULATION_CHECK is checked. In addition the integrity constraint US_POP_CHECK is enforced on US_CITIES since this

```
integrity US_POP_CHECK(CITIES:relation) is
begin
        for C in CITIES loop
                SUM:=SUM + C.POPULATION;
        end loop;
        if SUM > 225_000_000 then
                raise INTEGRITY_VIOLATION;
end
```

Figure 4-3: Integrity Declaration for a Relation Type

relation is of type CITIES. If this results in an integrity violation the violating operation on this relation is aborted and the exception INTEGRITY_VIOLATION is raised in the corresponding program block. Thus integrity constraints serve as background processes that are automatically activated by the system without explicit programmer defined intervention. Each time some alteration of a relation is done in the program the system scans the list of integrity constraints applicable to that relation. A usage of these constraints in a program is demonstrated in Fig. 4-4.

```
            ...
atomic transaction
    while not END_OF_FILE(INPUT_FILE) loop
            READ_IN(NEW_CITY);
            US_CITIES:=US_CITIES union NEW_CITY;
    end loop;
end atomic transaction
when INTEGRITY_VIOLATION =>
            PRINT("Overpopulation!");
            ...
```

Figure 4-4: Usage of a Constraint Exception in an Example Program

The use of integrity constraints can seriously degrade a programs efficiency. It is possible to suppress certain integrity checks within a program block by using the conventional Ada pragma feature. If we want to suppress the checking of the constraint US_POP_CHECK on the relation BIG_CITIES we could do so by specifying:

```
pragma SUPPRESS_INTEGRITY (US_POP_CHECK,on=>BIG_CITIES)
```

5. Example Applications Written in the Proposed System

In this section we will work out in detail a relatively complex example application written in our proposed language. In this example we make use of the Ada data abstraction facility, namely packages. This allows us to have a clear separation of the relation and procedure declarations from the actual code of the implementation.

Figure 5-1 defines a package for a typical university database that processes the grades of students. In this particular example we want to screen the students into two groups. Those whose [illegible] is lest 3.5 get

```
package GRADES is

type STUDENT_RELATION is
        relation (key STUDENT_NO) of
                STUDENT_NO:integer;
                NAME:string;
                ADVISOR:string;
        end relation

type COURSE_RELATION is
        relation (key COURSE_NO) of
                COURSE_NO:string;
                UNITS:0..12;
                PROFESSOR:string;
        end relation

type ENROLL_RELATION is
        relation (key STUDENT_NO,COURSE_NO) of
                COURSE_NO:string;
                STUDENT_NO:integer;
                GRADE:(A+,A,A-,B+,B,B-,C+,C,C-,D+,D,D-,F);
                TERM:string;
        end relation

type STUDENT_GRADES is
        relation (key STUDENT_NO) of
                STUDENT_NO:integer;
                NAME:string;
                GPA:0.0..4.0;
        end  relation

STUDENT:STUDENT_RELATION:=EXTERNAL_REL("STUDENT");     -- These relations
COURSE:COURSE_RELATION:=EXTERNAL_REL("COURSE");;       -- exist externally
ENROLL:ENROLL_RELATION:=EXTERNAL_REL("ENROLL");;       -- in the DBMS

GOOD,BAD:STUDENT_GRADES;

procedure SCREENING(out GOOD,BAD:STUDENT_GRADES);

procedure NOTIFY(in GOOD,BAD:STUDENT_GRADES);

procedure SEMESTER_END;
                .
                .  -- possibly more procedure declarations
                .

end package;
```

Figure 5-1: A Package Definition for a Student Database

Figure 5-1, continued

```
package body GRADES is

type COURSE_ENROLL is jointype (COURSE_RELATION,ENROLL_RELATION)
                on (COURSE_NO,COURSE_NO);

        procedure SCREENING(out GOOD,BAD:STUDENT_GRADES) is
        TOTGRADE:float;
        TOTUNITS:integer;
        GPA:0.0..4.0;
        CE_REL:COURSE_ENROLL;
        CE:recordtype COURSE_ENROLL;
        S:recordtype STUDENT_RELATION;
        begin
                CE_REL:=join (COURSE,ENROLL) on (COURSE_NO=COURSE_NO);
                for S in STUDENT loop
                        for CE in CE_REL where S.STUDENT_NO=CE.STUDENT_NO
                        loop
                           case CE.GRADE is
                                when A =>
                                   TOTGRADE:=TOTGRADE+4.0;
                                when A- =>
                                   TOTGRADE:=TOTGRADE+3.7;
                                        ...
                                        ...
                                when D =>
                                   TOTGRADE:=TOTGRADE+1.0;
                                when D- =>
                                   TOTGRADE:=TOTGRADE+0.7;
                           end case;
                           TOTUNITS:=TOTUNITS+CE.UNITS;
                        end loop;
                        GPA:=TOTGRADE/TOTUNITS;
                        if GPA >= 3.5 then
                                GOOD:=GOOD union (S.STUDENT_NO,S.NAME,GPA);
                        else
                                BAD:=BAD union (S.STUDENT_NO,S.NAME,GPA);
                end loop;
        end procedure SCREENING;

        procedure NOTIFY(in GOOD,BAD:STUDENT_GRADES) is
        S:recordtype STUDENT_GRADES;
        begin
                PRINT_TITLE("Students allowed to go on for Ph.D.:")
                for S in GOOD loop
                        PRINT_STUDENT(S.NAME,S.STUDENT_NO,S.GPA);
                end loop;

                PRINT_TITLE("Students not allowed to go on for Ph.D.:")
                for S in BAD loop
                        PRINT_STUDENT(S.NAME,S.STUDENT_NO,S.GPA);
                end loop;
        end --NOTIFY

        procedure SEMESTER_END is
        begin
                SCREENING(GOOD,BAD);
                NOTIFY(GOOD,BAD);
        end procedure SEMESTER_END;

end package body
```

stored in the relation GOOD, the others will be stored in the relation BAD. Subsequently they will be printed in two different output tables. The program makes use of the three external relations STUDENT, COURSE, and ENROLL. The explicit type definitions of these relations as well as of the relation type STUDENT_GRADES is given in the package declaration.

We will now describe the implementation of the procedure SCREENING which generates the two relations GOOD and BAD, which can subsequently be used for further processing in the package. But since these two relations are not *external* they do not get stored permanently in the DBMS. The package body shows the actual code for this procedure. First we create one relation CE_REL which is the join of the two relations COURSE and ENROLL on identical COURSE_NO fields. Thus CE_REL has the six fields COURSE_NO, UNITS, PROFESSOR, STUDENT_NO, GRADE, and TERM. Then we loop through the STUDENT relation and determine for each student in turn the values TOTGRADE and TOTUNITS by looping through the tuples of the relation CE_REL which have that particular value for the STUDENT_NO field. Subsequently the GPA is computed. Depending on the value of the gpa a new tuple is appended (via union) to the relation GOOD or BAD.

The procedure NOTIFY then makes use of the two relations GOOD and BAD in order to generate an output listing of the two groups of students, in this case those who can continue for a Ph.D. degree and those who cannot. In this procedure we make use of the two output functions PRINT_TITLE and PRINT_STUDENT which we assume to be defined elsewhere (Their implementation is straightforward and does not illustrate any aspects of the relational capabilities of AdaRel. Therefore it is omitted here.).

The procedure SEMESTER_END merely calls the two procedures SCREENING and NOTIFY with the appropriate parameters. Thus the whole process can get started by calling GRADES.SEMESTER_END from outside the package. It results in an output listing of the good and bad students.

6. Conclusions

This paper is concerned with the insertion of relations, relational operators, concurrency control, exception handling for database programming, and integrity control into Ada. For matter of presentation we have described these features at the user level and have avoided detailed descriptions of each feature. Our goal is twofold: (i) to give an appreciation of the added power of a programming language when such features are included; and (ii) to show that the new programming language, AdaRel, can incorporate such features without requiring any changes in Ada's underlying semantics. Further issues remain which we do not discuss here. One major issue is the question of implementation. In a later paper we will show how these features can be translated into Ada making only reasonable assumptions on the Ada compiler and the associated DBMS. This approach was taken by Stonebraker et.al. in implementing Equel [20]. Another issue is efficiency. Ultimately this issue cannot be decided until a prototype is actually constructed.

Ada is already a big and comprehensive programming language. However, it is still very much an imperative programming language in the tradition of Pascal and Algol60. Our attempt to inject relations and relational language constructs raises the expressive power of Ada in a major way. Though several relational data manipulation have been implemented without an underlying general purpose programming language, this has lead to a lack of flexibility and an inability to gain program efficiency. The coupling of Ada and a DBMS through the definition of relation types and relational operators achieves a highly expressive tool which can increase programmer productivity substantially.

References

[1] United States Department of Defense.
Reference Manual for the Ada Programming Language.
1980.

[2] Astrahan,M.M. et al.
System R: Relational Approach to Database Management.
ACM-TODS 1(2):97-137, June, 1976.

[3] Chamberlain,D.D., Gray,J.N., and Traiger,I.L.
Views, Authorization, and Locking in a Relational Database System.
In *Proceedings of the National Computer Conference*, pages 425-430. AFIPS, 1975.

[4] Codd,E.F.
The 1981 Turing Award Lecture. Relational Database:A Practical Foundation for Productivity.
CACM 25(2), Feb, 1982.

[5] Date,C.J.
An Introduction to Database Systems.
Addison-Wesley Publishing Company, 1977.

[6] Information Builders, Inc.
FOCUS Users Manual.
Information Builders, Inc. 1250 Broadway, New York, N.Y.10001, 1982.

[7] M.Hammer,W.Howe,V.Kruskal,and I.Wladawsky.
A Very High Level Programming Language for Data Processing Applications.
CACM 20(11):832-840, Nov, 1977.

[8] Horowitz,E., Kemper,A., and Narasimhan,B.
An Analysis of Application Generators.
Technical Report TR-83-208, USC, March, 1983.
submitted for publication to IEEE Transaction on Software Engineering.

[9] Martin,J.
Application Development Without Programmers.
Prentice Hall,Inc., Englewood Cliffs, New Jersey, 1982.

[10] McCracken,D.D.
A Guide To Nomad For Application Development.
National CSS, 1980.

[11] National CSS.
NOMAD 2: Reference Manual.
National CSS. Wilton, CT 06897, 1982.

[12] MPG.
RAMIS 2: Users manual.
Mathematica Products Group, Inc. Princeton, N.J. 08540, 1982.

[13] Rowe,L.A. and Shoens,K.A.
Data Abstractions, Views and Updates in Rigel.
In Bernstein,P.A. (editor), *Proceedings International Conference on Management of Data*, pages 71-81. ACM-SIGMOD, May, 1979.

[14] Rowe,L.A. and Shoens,K.A.
Programming Language Constructs for Screen Definition.
IEEE Transactions on Software Engineering SE-9(1):31-39, Jan, 83.

[15] Schmidt,J.W.
Some High Level Language Constructs for Data of Type Relation.
ACM-TODS 2(3), Sept, 1977.

[16] Schmidt,J.W.
Type Concepts for Database Definition.
In *Proc. Int. Conf. on Databases:Improving Usability and Responsiveness*, pages 215-244.
Academic Press,New York, 1978.

[17] Shipman,D.
The Functional Data Model and the Data Language DAPLEX.
ACM-TODS 6(1):140-173, March, 1981.

[18] Shopiro,J.E.
THESEUS-A Programming Language for Relational Databases.
ACM-TODS 4(4):493-517, December, 1979.

[19] Smith,J., Fox,S. and Landers,T.
Reference Manual for ADAPLEX.
Computer Corporation of America, 1981.

[20] Stonebraker,M., et al.
The Design and Implementation of Ingres.
TODS 1(3), Sep, 1976.

[21] Ullman,J.
Principles of Database Systems.
Computer Science Press, Potomac, Maryland, 1980.

[22] Wasserman, A.I. et al.
Revised Report on the Programming Language PLAIN.
ACM-SIGPLAN , May, 1981.